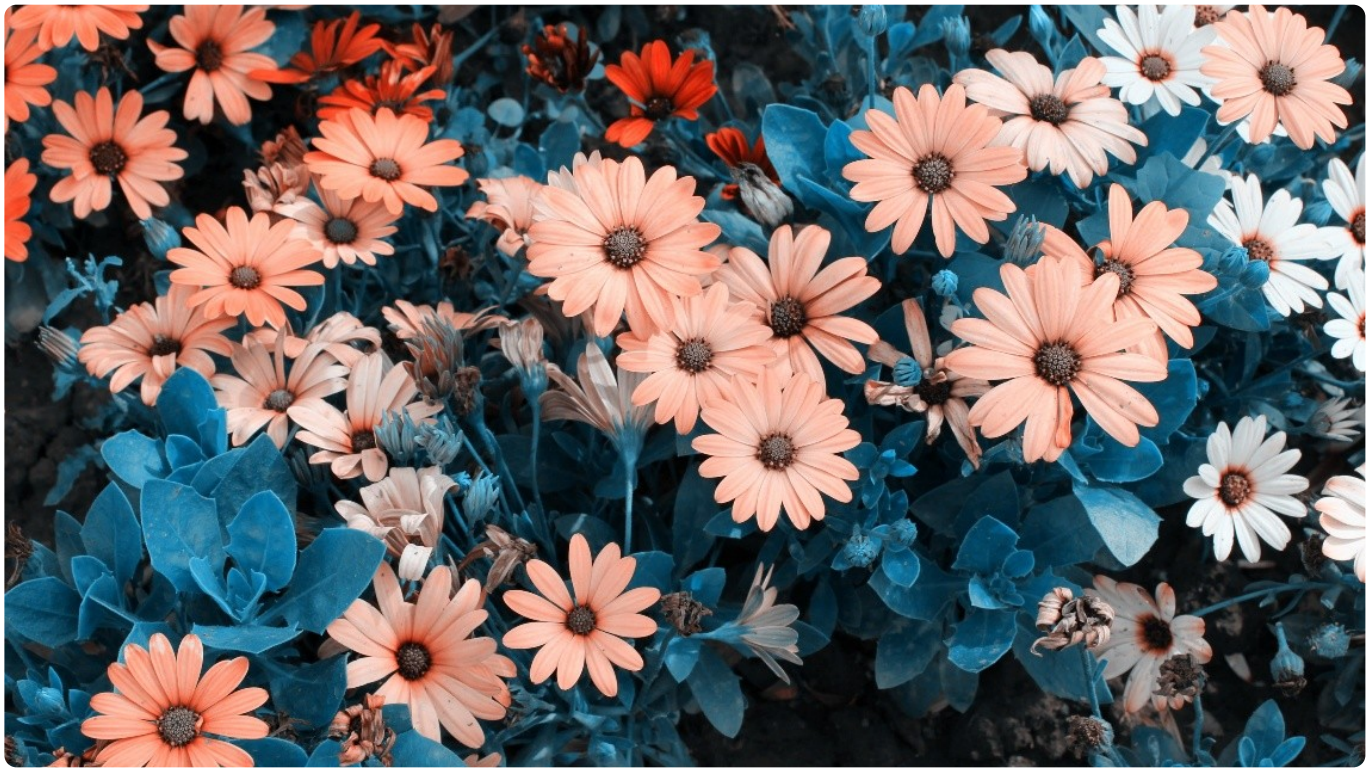
03 | 语法分析：两个基本功和两种算法思路

2020-06-05 宫文学



讲述：宫文学
时长 20:16 大小 18.57M

你好，我是宫文学。

通过第 1 讲的学习，现在你已经清楚了语法分析阶段的任务：依据语法规则，把 Token 串转化成 AST。

今天，我就带你来掌握语法分析阶段的核心知识点，也就是两个基本功和两种算法思路。理解了这些重要的知识点，对于语法分析，你就不是外行了。

两个基本功：第一，必须能够阅读和书写语法规则，也就是掌握上下文无关文法；第二，必须要掌握递归下降算法。

两种算法思路：一种是自顶向下的语法分析，另一种则是自底向上的语法分析。

上下文无关文法（Context-Free Grammar）

在开始语法分析之前，我们要解决的第一个问题，就是**如何表达语法规则**。在上一讲中，你已经了解了，我们可以用正则表达式来表达词法规则，语法规则其实也差不多。

我还是以下面这个示例程序为例，里面用到了变量声明语句、加法表达式，我们看看语法规则应该怎么写：

复制代码

```
int a = 2;
int b = a + 3;
return b;
```

第一种写法是下面这个样子，它看起来跟上一讲的词法规则差不多，都是左边是规则名称，右边是正则表达式。

复制代码

```
start: blockStmts ;                 //起始
block : '{' blockStmts '}' ;        //语句块
blockStmts : stmt* ;                //语句块中的语句
stmt = varDecl | expStmt | returnStmt | block;    //语句
varDecl : type Id varInitializer?  ';' ;          //变量声明
type : Int | Long ;                               //类型
varInitializer : '=' exp ;                        //变量初始化
expStmt : exp ';' ;                               //表达式语句
returnStmt : Return exp ';' ;                     //return语句
exp : add ;                                       //表达式
add : add '+' mul | mul;                          //加法表达式
mul : mul '*' pri | pri;                          //乘法表达式
pri : IntLiteral | Id | '(' exp ')' ;             //基础表达式
```

在语法规则里，我们把冒号左边的叫做**非终结符**（Non-terminal），又叫**变元**（Variable）。非终结符可以按照右边的正则表达式来逐步展开，直到最后都变成标识符、字面量、运算符这些不可再展开的符号，也就是**终结符**（Terminal）。终结符其实也是词法分析过程中形成的 Token。

提示：

1. 在本课程，非终结符以小写字母开头，终结符则以大写字母开头，或者是一个原始的字

符串格式。
2. 在谈论语法分析的时候，我们可以把 Token 和非终结符这两个术语互换使用。

像这样左边是非终结符，右边是正则表达式的书写语法规则的方式，就叫做**扩展巴科斯范式（EBNF）**。你在 ANTLR 这样的语法分析器生成工具中，经常会看到这种格式的语法规则。

对于 EBNF 的严格定义，你可以去参考Wikipedia上的解释。

在教科书中，我们还经常采用另一种写法，就是**产生式**（Production Rule），又叫做**替换规则**（Substitution Rule）。产生式的左边是非终结符（变元），它可以用右边的部分替代，中间通常会用箭头连接。

为了避免跟 EBNF 中的“*”号、“+”号等冲突，在本节课中，凡是采用 EBNF 格式，就给字符串格式的终结符加引号，左右两边用“::=”或冒号分隔开；凡是采用产生式，字符串就不加引号，并且采用“->”分隔产生式的左右两侧。

复制代码

```
add -> add + mul
add -> mul
mul -> mul * pri
mul -> pri
```

也有个偷懒的写法，就是把同一个变元的多个产生式写在一起，用竖线分隔（但这时候，如果产生式里面原本就要用到“|”终结符，那么就要加引号来进行区分）。但也就仅此为止了，不会再引入“*”和“+”等符号，否则就成了 EBNF 了。

复制代码

```
add -> add + mul | mul
mul -> mul * pri | pri
```

产生式不用“ * ”和“+”来表示重复，而是用迭代，并引入“ε”（空字符串）。所以“blockStmts : stmt*”可以写成下面这个样子：

```
blockStmts -> stmt blockStmts | ε
```

复制代码

总结起来，语法规则是由 4 个部分组成的：

一个有穷的非终结符（或变元）的集合；

一个有穷的终结符的集合；

一个有穷的产生式集合；

一个起始非终结符（变元）。

那么符合这四个特点的文法规则，就叫做**上下文无关文法**（Context-Free Grammar，CFG）。

你可能会问，**上下文无关文法和词法分析中用到的正则文法是否有一定的关系？**

是的，正则文法是上下文无关文法的一个子集。其实，正则文法也可以写成产生式的格式。比如，数字字面量（正则表达式为“[0-9]+”）可以写成：

复制代码

```
IntLiteral -> Digit IntLiteral1
IntLiteral1 -> Digit IntLiteral1
IntLiteral1 -> ε
Digit -> [0-9]
```

但是，在上下文无关文法里，产生式的右边可以放置任意的终结符和非终结符，而正则文法只是其中的一个子集，叫做**线性文法**（Linear Grammar）。它的特点是产生式的右边部分最多只有一个非终结符，比如 X->aYb，其中 a 和 b 是终结符。

图 1：正则文法是上下文无关文法的子集

你可以试一下，把上一讲用到的正则表达式“a[a-zA-Z0-9]*bc”写成产生式的格式，它就符合线性文法的特点。

```
1 S0 -> aS1bc
2 S1 -> [a-zA-Z0-9]S1
3 S1 -> ε
```

但对于常见的语法规则来说，正则文法是不够的。比如，你最常用的算术表达式的规则，就没法用正则文法表示，因为有的产生式需要包含两个非终结符（如“add + mul”）。你可以试试看，能把“2+3”“2+3*5”“2+3+4+5”等各种可能的算术表达式，用一个正则表达式写出来吗？实际是不可能的。

```
1 add -> add + mul
2 add -> mul
3 mul -> mul * pri
4 mul -> pri
```

好，现在你已经了解了上下文无关文法，以及它与正则文法的区别。可是，**为什么它会叫“上下文无关文法”这样一个奇怪的名字呢？难道还有上下文相关的文法吗？**

答案的确是有的。举个例子来说，在高级语言里，本地变量必须先声明，才能在后面使用。这种制约关系就是上下文相关的。

不过，在语法分析阶段，我们一般不管上下文之间的依赖关系，这样能使得语法分析的任务更简单。而对于上下文相关的情况，则放到语义分析阶段再去处理。

好了，现在你已经知道，用上下文无关文法可以描述程序的语法结构。学习编译原理，阅读和书写语法规则是一项基本功。针对高级语言中的各种语句，你要都能够手写出它们的语法规则来才可以。

接下来，我们就要**依据语法规则，编写语法分析程序，把 Token 串转化成 AST。**语法分析的算法有很多，但有一个算法也是你必须掌握的一项基本功，这就是递归下降算法。

递归下降算法（Recursive Descent Parsing）

递归下降算法其实很简单，它的基本思路就是按照语法规则去匹配 Token 串。比如说，变量声明语句的规则如下：

复制代码

```
varDecl : types Id varInitializer?  ';' ;         //变量声明
varInitializer : '=' exp ;                        //变量初始化
exp : add ;                                       //表达式
add : add '+' mul | mul;                          //加法表达式
mul : mul '*' pri | pri;                          //乘法表达式
pri : IntLiteral | Id | '(' exp ')' ;             //基础表达式
```

如果写成产生式格式，是下面这样：

复制代码

```
varDecl -> types Id varInitializer ';'
varInitializer -> '=' exp
varInitializer -> ε
exp -> add
add -> add + mul
add -> mul
mul -> mul * pri
mul -> pri
pri -> IntLiteral
pri -> Id
pri -> ( exp )
```

而基于这个规则做解析的算法如下：

```
匹配一个数据类型(types)
匹配一个标识符(Id)，作为变量名称
匹配初始化部分(varInitializer)，而这会导致下降一层，使用一个新的语法规则：
    匹配一个等号
    匹配一个表达式(在这个步骤会导致多层下降：exp->add->mul->pri->IntLiteral)
    创建一个varInitializer对应的AST节点并返回
如果没有成功地匹配初始化部分，则回溯，匹配ε，也就是没有初始化部分。
匹配一个分号
创建一个varDecl对应的AST节点并返回
```

用上述算法解析“int a = 2”，就会生成下面的 AST：

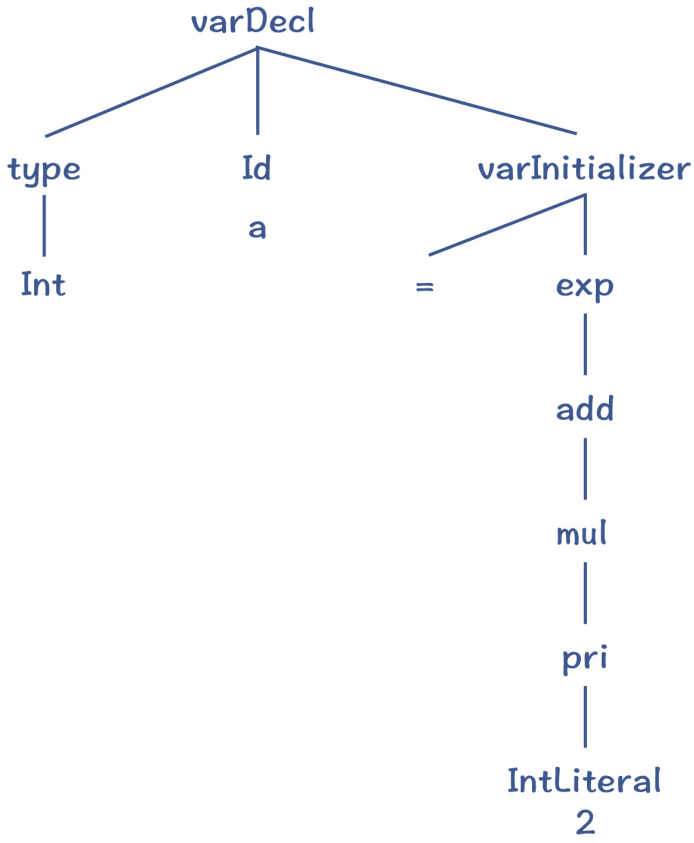


图 2：“int a = 2”对应的 AST

那么总结起来，递归下降算法的特点是：

对于一个非终结符，要从左到右依次匹配其产生式中的每个项，包括非终结符和终结符。

在匹配产生式右边的非终结符时，要下降一层，继续匹配该非终结符的产生式。

如果一个语法规则有多个可选的产生式，那么只要有一个产生式匹配成功就行。如果一个产生式匹配不成功，那就回退回来，尝试另一个产生式。这种回退过程，叫做**回溯**（Backtracking）。

所以说，递归下降算法是非常容易理解的。它能非常有效地处理很多语法规则，但是它也有两个缺点。

第一个缺点，就是著名的左递归（Left Recursion）问题。比如，在匹配算术表达式时，产生式的第一项就是一个非终结符 add，那么按照算法，要下降一层，继续匹配 add。这个过程会一直持续下去，无限递归下去。

```
1  add -> add + mul
```

复制代码

所以，递归下降算法是无法处理左递归问题的。那么有什么解决办法吗？

你可能会说，把产生式改成右递归不就可以了吗？也就是 add 这个递归项在右边：

```
1  add -> mul + add
```

复制代码

这样确实可以避免左递归问题，但它同时也会导致**结合性**的问题。

举个例子来说，我们按照上面的语法规则来解析“2+3+4”这个表达式，会形成如下所示的 AST。

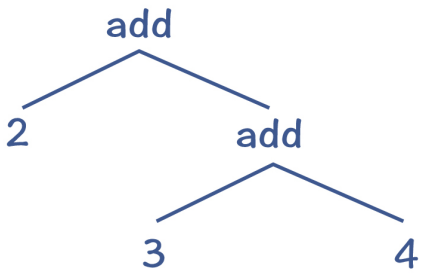


图 3：结合性错误的 AST

它会先计算“3+4”，而不是先计算“2+3”。这破坏了加法的结合性规则，加法运算本来应该是左结合的。

其实有一个标准的方法，能避免左递归问题。我们可以改写原来的语法规则，也就是引入 add'，把左递归变成右递归：

复制代码

```
add -> mul add'
add' -> + mul add' | ε
```

接下来，我们用刚刚改写的规则再次解析一下 “2+3+4”这个表达式，会得到下图中的 AST：

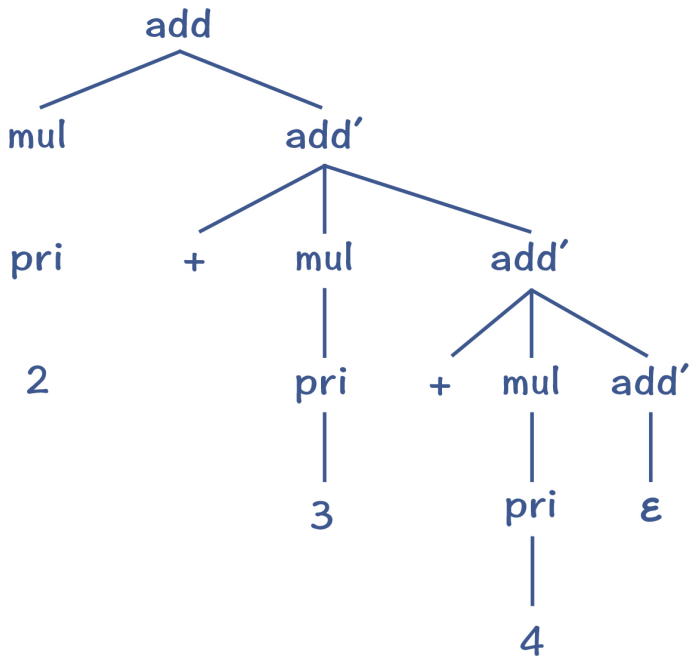


图 4：基于改写后的文法所生成的 AST

你能看出，这种改写方法虽然能够避免左递归问题，但由于add'的规则是右递归的，采用标准的递归下降算法，仍然会出现运算符结合性的错误。那么针对这点，我们有没有解决办法呢？

有的，方法就是**把递归调用转化成循环**。这里利用了很多同学都知道的一个原理，即递归调用可以转化为循环。

其实我把上面的规则换成用 EBNF 方式来表达就很清楚了。在 EBNF 格式里，允许用“*”号和“+”号表示重复：

复制代码

```
add :  mul ('+' mul)*  ;
```

所以说，对于('+'mul)*这部分，我们其实可以写成一个循环。而在循环里，我们可以根据结合性的要求，手工生成正确的 AST。它的伪代码如下：

复制代码

```
左子节点 = 匹配一个mul
while(下一个Token是+){
   消化掉+
   右子节点 = 匹配一个mul
   用左、右子节点创建一个add节点
   左子节点 = 该add节点
}
```

采用上面的算法，就可以创建正确的 AST，如下图所示：

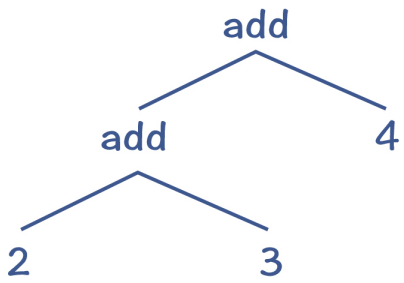


图 5：结合性正确的 AST

递归下降算法的第二个缺点，就是当产生式匹配失败的时候，必须要“回溯”，这就可能导致浪费。

这个时候，我们有个针对性的解决办法，就是预读后续的一个 Token，判断该选择哪个产生式。

以 stmt 变元为例，考虑它的三个产生式，分别是变量声明语句、表达式语句和 return 语句。那么在递归下降算法中，我们可以在这里预读一个 Token，看看能否根据这个 Token 来选择某个产生式。

经过仔细观察，你发现如果预读的 Token 是 Int 或 Long，就选择变量声明语句；如果是 IntLiteral、Id 或左括号，就选择表达式语句；而如果是 Return，则肯定是选择 return 语句。因为这三个语句开头的 Token 是不重叠的，所以你可以很明确地做出选择。

如果我们手写递归下降算法，可以用肉眼识别出每次应该基于哪个 Token，选择用哪个产生式。但是，对于一些比较复杂的语法规则，我们要去看好几层规则，这样比较辛苦。

那么能否有一个算法，来自动计算出选择不同产生式的依据呢？当然是有的，这就是 LL 算法家族。

LL 算法：计算 First 和 Follow 集合

LL 算法的要点，就是计算 First 和 Follow 集合。

First 集合是每个产生式开头可能会出现的 Token 的集合。就像 stmt 有三个产生式，它的 First 集合如下表所示。

非终结符（变元）	First集合
varDecl	Int Long
expStmt	IntLiteral Id (
returnStmt	Return
stmt	Int Long IntLiteral Id (Return

而 stmt 的 First 集合，就是三个产生式的 First 集合的并集，也是 Int Long IntLiteral Id (Return。

总体来说，针对非终结符 x，它的 First 集合的计算规则是这样的：

如果产生式以终结符开头，那么把这个终结符加入 First(x)；

如果产生式以非终结符 y 开头，那么把 First(y) 加入 First(x);

如果 First(y) 包含ε，那要把下一个项的 First 集合也加入进来，以此类推；

如果 x 有多个产生式，那么 First(x) 是每个产生式的并集。

在计算 First 集合的时候，具体可以采用“**不动点法**”。相关细节这里就不展开了，你可以参考示例程序FirstFollowSet类的 CalcFirstSets() 方法，运行示例程序能打印各个非终结符的 First 集合。

不过，这样是不是就万事大吉了呢？

其实还有一种特殊情况我们需要考虑，那就是对于某个非终结符，它自身会产生ε的情况。比如说，示例文法中的 blockStmts，它是可能产生ε的，也就是块中一个语句都没有。

复制代码

```
block : '{' blockStmts '}' ;                //语句块
blockStmts : stmt* ;                        //语句块中的语句
stmt = varDecl | expStmt | returnStmt;      //语句
```

语法解析器在这个时候预读的下一个 Token 是什么呢？是右花括号。这证明 blockStmts 产生了ε，所以才读到了后续跟着的花括号。

对于某个非终结符后面可能跟着的 Token 的集合，我们叫做 Follow 集合。如果预读到的 Token 在 Follow 中，那么我们就可以判断当前正在匹配的这个非终结符，产生了ε。

Follow 的算法也比较简单，以非终结符 x 为例：

扫描语法规则，看看 x 后面都可能跟着哪些符号；

对于后面跟着的终结符，都加到 Follow(x) 集合中去；

如果后面是非终结符 y，就把 First(y) 加 Follow(x) 集合中去；

最后，如果 First(y) 中包含ε，就继续往后找；

如果 x 可能出现在程序结尾，那么要把程序的终结符 $ 加入到 Follow(x) 中去。

这样在计算了 First 和 Follow 集合之后，你就可以通过预读一个 Token，来完全确定采用哪个产生式。这种算法，就叫做 **LL(1) 算法**。

LL(1) 中的第一个 L，是 Left-to-right 的缩写，代表从左向右处理 Token 串。第二个 L，是 Leftmost 的缩写，意思是最左推导。**最左推导是什么呢？**就是它总是先把产生式中最左侧的非终结符展开完毕以后，再去展开下一个。这也就相当于对 AST 从左子节点开始的深度优先遍历。LL(1) 中的 1，指的是预读一个 Token。

LR 算法：移进和规约

前面讲的递归下降和 LL 算法，都是自顶向下的算法。还有一类算法，是自底向上的，其中的代表就是 **LR 算法**。

自顶向下的算法，是从根节点逐层往下分解，形成最后的 AST；而 LR 算法的原理呢，则是从底下先拼凑出 AST 的一些局部拼图，并逐步组装成一棵完整的 AST。**所以，其中的关键之处在于如何“拼凑”。**

假设我们采用下面的上下文无关文法，来推演一个实例，具体语法规则如下所示：

```
start->add
add->add+mul
add->mul
mul->mul*pri
mul->pri
pri->Int
pri->(add)
```

如果用于解析“2+3*5”，最终会形成下面的 AST：

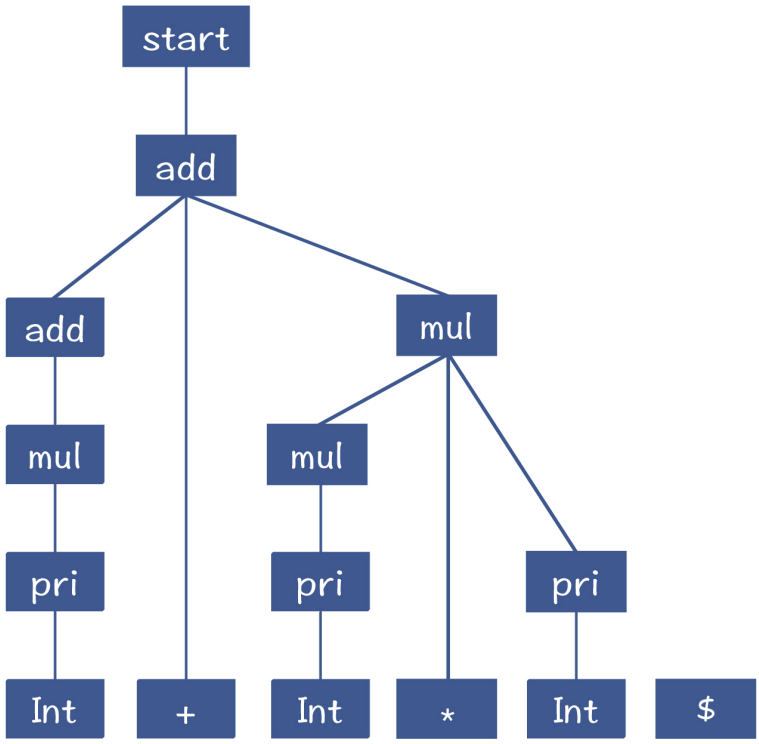


图 6：2+3*5 对应的 AST

那算法是怎么从底部凑出这棵 AST 来的呢？

LR 算法和 LL 算法一样，也是从左到右地消化掉 Token。在第 1 步，它会取出“2”这个 Token，放到一个栈里，这个栈是用来组装 AST 的工作区。同时，它还会预读下一个 Token，也就是“+”号，用来帮助算法做判断。

在下面的示意图里，我画了一条橙色竖线，竖线的左边是栈，右边是预读到的一个 Token。在做语法解析的过程中，竖线会不断地往右移动，把 Token 放到栈里，这个过程叫做“**移进**”（Shift）。

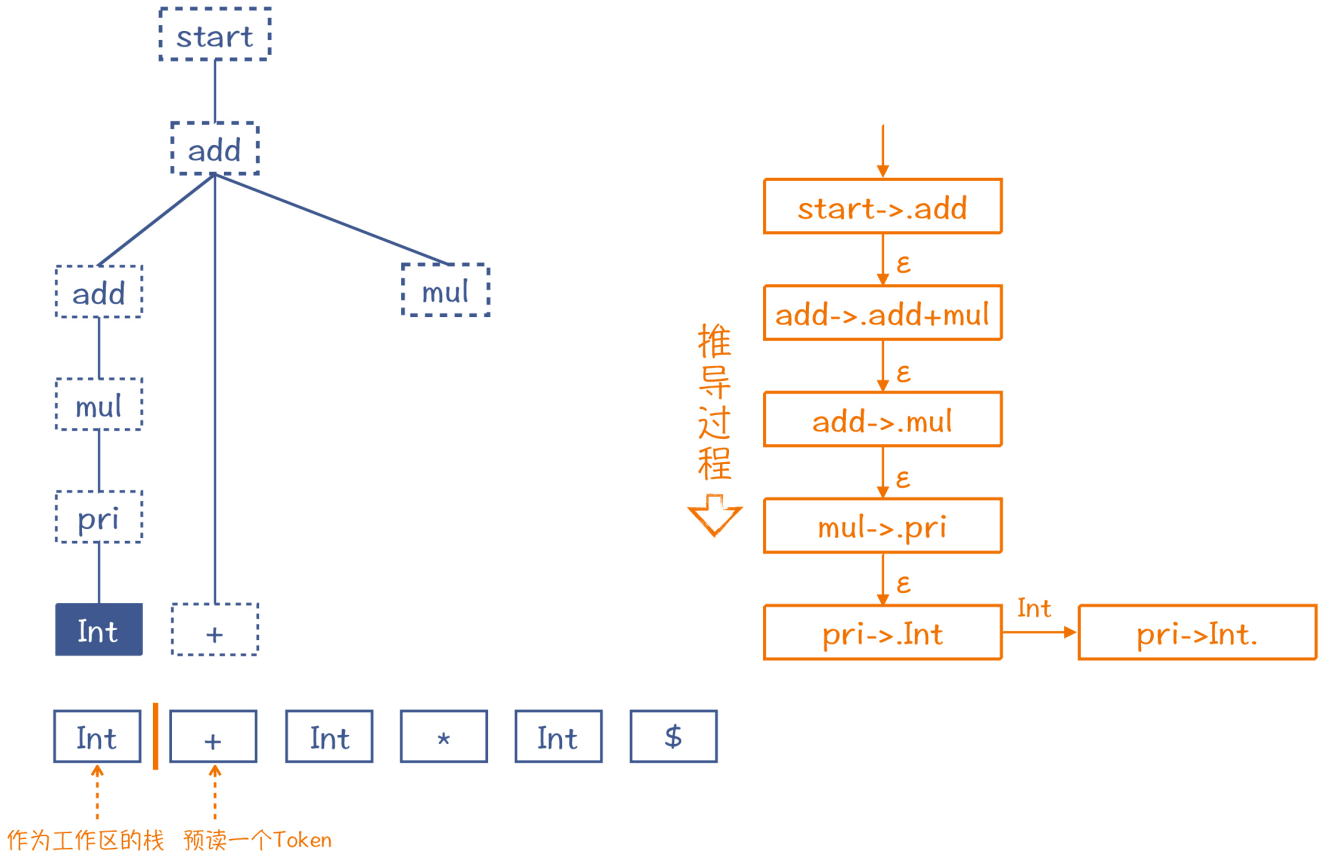


图 7：第 1 步，移进一个 Token

注意，我在图 7 中还用虚线框推测了 AST 的其他部分。也就是说，如果第一个 Token 遇到的是整型字面量，而后面跟着一个 + 号，那么这两个 Token 就决定了它们必然是这棵推测出来的 AST 的一部分。而图中右边就是它的推导过程，其中的每个步骤，都使用了一个产生式加了一个点（如“.add”）。这个点，就相当于图中左边的橙色竖线。

所以你就可以根据这棵假想的 AST，也就是依据假想的推导过程，给它反推回去。把 Int 还原为 pri。这个还原过程，就叫做“**规约**”（Reduce）。工作区里的元素也随之更新成 pri。

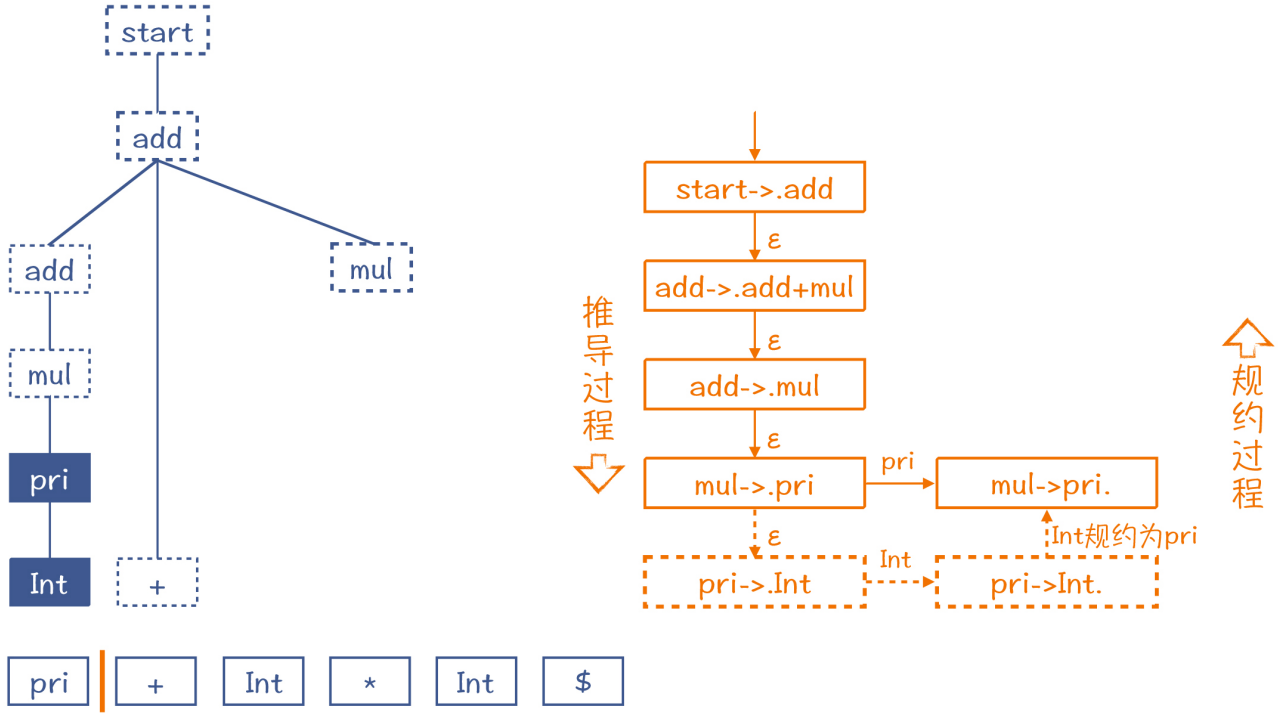


图 8：第 2 步，Int 规约为 pri

按照这样的思路，不断地移进和规约，这棵 AST 中推测出来的节点会不断地被证实。而随着读入的 Token 越来越多，这棵 AST 也会长得越来越高，整棵树变得更大。下图是推导过程中间的一个步骤。

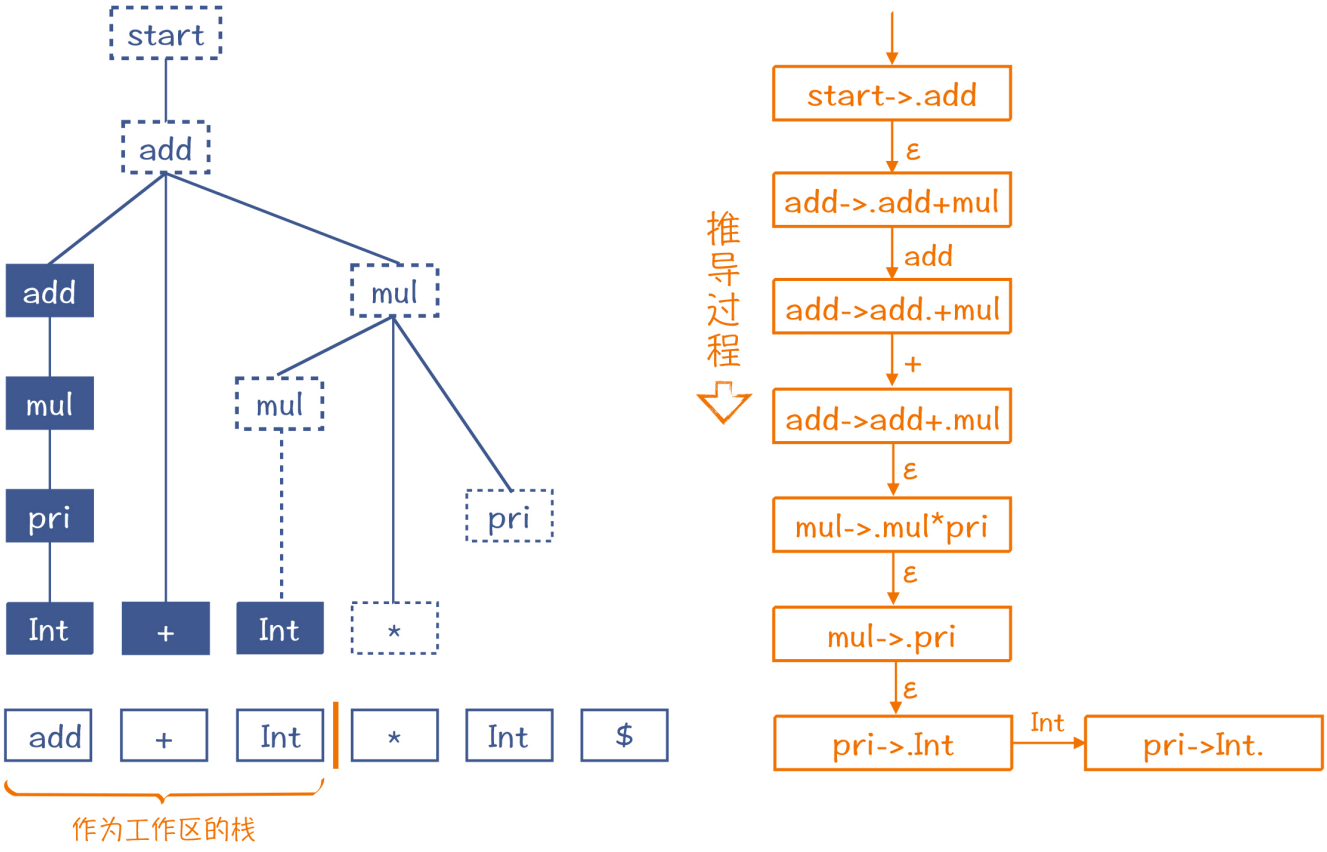


图 9：移进和规约过程中的一个步骤

最后，整个 AST 构造完毕，而工作区里也就只剩了一个 Start 节点。

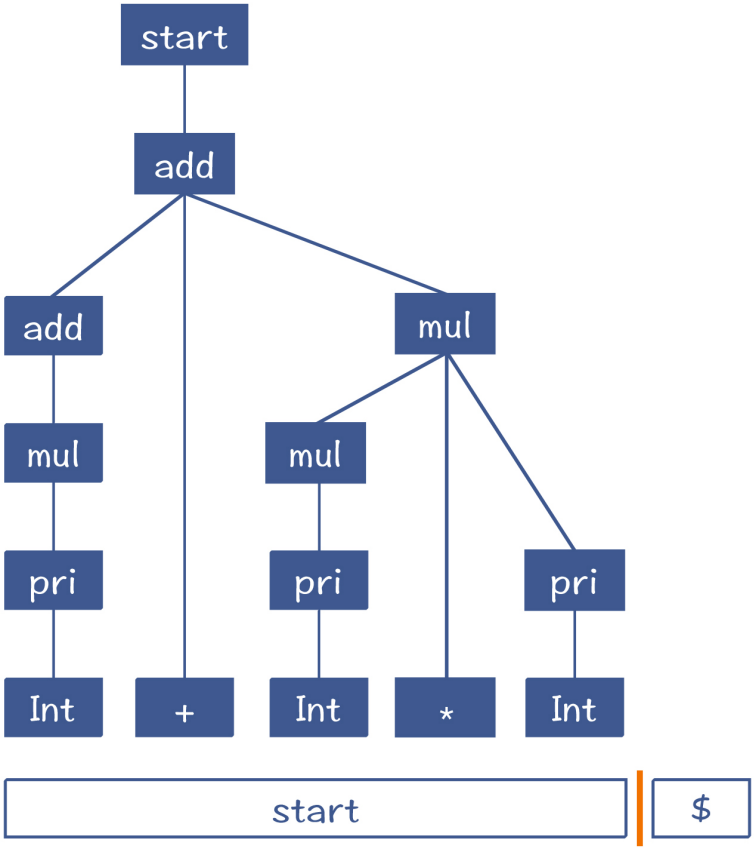


图 10：最后一步，add 规约为 start

通过上面的介绍，你应该已经建立了对 LR 算法的直觉认识。如果要把这个推导过程写成严密的算法，你可以参考《编译原理之美》的第 18 讲。

从示例中，你应该已经看出来了，相对于 LL 算法，LR 算法的优点是能够处理左递归文法。但它也有缺点，比如不利于输出全面的编译错误信息。因为在没有解析完毕之前，算法并不知道最后的 AST 是什么样子，所以也不清楚当前的语法错误在整体 AST 中的位置。

最后我再提一下 LR 的意思，来帮你更完整地理解 LR 算法。L 还是代表从左到右读入 Token，而 R 是最右推导（Rightmost）的意思。我把“2+3*5”最右推导的过程写在了下面，而如果你从最后一行往前一步步地看，它恰好就是规约的过程。

```
start -> add
      -> add + mul
      -> add + mul * pri
      -> add + mul * Int
      -> add + pri * Int
      -> add + Int * Int
      -> mul + Int * Int
      -> pri + Int * Int
      -> Int + Int * Int
```

如果你见到 LR(k)，那它的意思就是会预读 k 个 Token，我们在示例中采用的是 LR(1)。

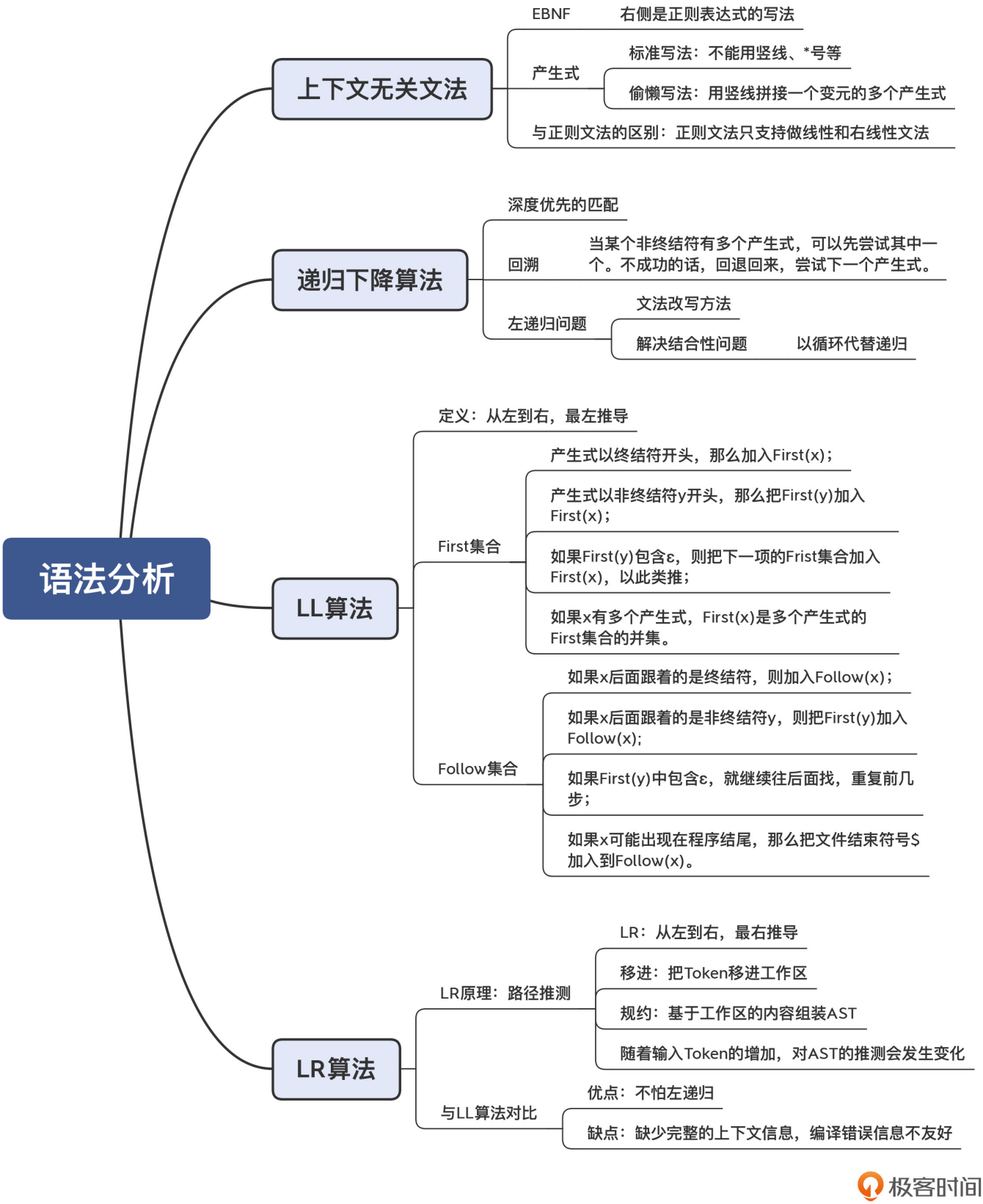
课程小结

今天花了一讲的时间，把语法分析的要点给你讲解了一下。

对于上下文无关的文法，你要知道产生式、非终结符、终结符、EBNF 这几个基本概念，能够熟练阅读各种语言的语法规则，这是一个基本功。

递归下降算法是另一项基本功，所以也一定要掌握。**你要注意，递归下降是深度优先的，只有最左边的子树都生成完了，才会往右生成它的兄弟节点。**有的同学会在没有把左侧的非终结符匹配完毕的情况下，就开始匹配右边的项，从而不自觉地采用了宽度优先的思路，这是我发现很多同学会容易陷入的一个思维误区。

对于 LL 算法和 LR 算法，我只做了简单的讲解，目的是为了帮助你建立直观的理解。我们在后面的课程中，还会遇到使用它们的实际例子，到时你可以与这一讲的内容相互印证。



一课一思

你可以计算一下示例文法中 block、blockStmts、stmt、varDecl、returnStmt 和 expStmt 的 First 和 Follow 集合吗？这样，你也可以熟悉一下 First 和 Follow 集合的计算方法。

欢迎在留言区分享你的答案。如果觉得有收获，也欢迎你把这节课分享给你的朋友。

参考资料

1. 线性文法（Linear Grammar）：参见Wikipedia。

2. 左递归及其消除方法：参见Wikipedia。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　02 | 词法分析：用两种方式构造有限自动机

下一篇　04 | 语义分析：让程序符合语义规则

精选留言 (6)

写留言

Apsaras
2020-06-05

老师，递归下降算法、LL(1)算法、LR算法的相互之间是替代关系吗？还是LL(1)和LR都是递归下降的补充？

作者回复: 递归下降算法和LL算法都是自顶向下的。递归下降强调的是用“递归调用”和“层层下降”的方式来书写代码。所以LL算法可以用递归下降的方式编写（Java编译器），也可以用查表的方式编写（Python编译器），这两种方式你在后面的课程里都可以看到。
LR算法是完全不同的，是自底向上的算法。

1 2

thomas1994
2020-06-08

赞，老师的实战课程比编译原理之美更加通俗易懂

展开

作者回复: 谢谢肯定！
说明老师也有进步:-)

1

Charles
2020-06-09

开始的地方提到算术表达式无法用正则表达式匹配，好像不对吧？
我试了一下，[0-9]+((\+[0-9]+)*(*[0-9]+)*)*就可以匹配。

击水湘江
2020-06-09

参考资料可以再丰富些不？初次接触编译原理，感觉看完一头雾水😭

作者回复: 好，我尽量多补充点链接！

易昊
2020-06-09

我的计算结果是：
FIRST(block) = {'{'}
FIRST(blockStmts) = {'{', ε}
FIRST(stmt) = {Int, Long, IntLiteral, Id, '(', return, '{'}
FIRST(varDecl) = {Int, Long}...

展开

SIGHOR

2020-06-05

慢慢的恢复大学里学习这门课的感觉，需要再多看几遍。
而且我突然发现有点不太会学习了

展开

作者回复: 嗯，肯定要多看几遍。因为里面知识点还是挺密集的。